MORRIS MWANGI INTE/MG/3110/09/22

Function is f(x) = x² - x - 2

Guesses: a = 1, b = 3

Iteration 1:

f(a) = f(1) = 1² - 1 - 2 = -2

f(b) = f(3) = 3² - 3 - 2 = 4

x = (af(b) - bf(a)) / (f(b) - f(a))

= (14 - 3(-2)) / (4 - (-2))

= 10 / 6

≈ 1.6667

f(x) = 1.6667² - 1.6667 - 2 ≈ -0.0556

Since f(a) \_ f(x) < 0, b = x

Iteration 2:

a = 1, b = 1.6667

f(a) = -2

f(b) = -0.0556

x = (1\_(-0.0556) - 1.6667\*(-2)) / ((-0.0556) - (-2))

≈ 1.5616

f(x) = 1.5616² - 1.5616 - 2 ≈ -0.0008

Since f(a) \* f(x) < 0, b = x

Iteration 3:

a = 1, b = 1.5616

f(a) = -2

f(b) = -0.0008

x = (1\*(-0.0008) - 1.5616\*(-2)) / ((-0.0008) - (-2))

≈ 1.5538

f(x) = 1.5538² - 1.5538 - 2 ≈ -0.00001

The final approximation is x ≈ 1.5538.

import numpy as np

import matplotlib.pyplot as plt

x = np.array([2.00, 4.25, 5.25, 7.81, 9.20, 10.60])

y = np.array([7.2, 7.1, 6.0, 5.0, 3.5, 5.0])

def linear-interpolation(x0, x1, y0, y1, x):

"""Linear interpolation between two points."""

return y0 + (x - x0) \* (y1 - y0) / (x1 - x0)

i = np.searchsorted(x, 4.0) - 1

x0, x1 = x[i], x[i + 1]

y0, y1 = y[i], y[i + 1]

y\_interpolated = linear\_interpolation(x0, x1, y0, y1, 4.0)

print(f"The interpolated y-value at x=4.0 is: {y\_interpolated:.4f}")

plt.figure(figsize=(10, 6))

plt.plot(x, y, "bo-", label="Given points")

plt.plot([4.0], [y\_interpolated], "ro", label="Interpolated point")

plt.plot([x0, x1], [y0, y1], "g-", label="Interpolation segment")

plt.xlabel("X (in)")

plt.ylabel("Y (in)")

plt.title("Linear Interpolation for Robot Laser Scanner")

plt.legend()

plt.grid(True)

plt.savefig("./assets/c-linear\_interpolation.png")

The equation is: f(x) = x³ - 0.165x² + 3.993 × 10⁻⁴

The derivative is: f'(x) = 3x² - 0.33x

The initial guess is: (x₀)

Newton's method formula is: xₙ₊₁ = xₙ - f(xₙ) / f'(xₙ)

Let x₀ = 0.05

First iteration:

f(0.05) = 0.05³ - 0.165(0.05²) + 3.993 × 10⁻⁴ = 0.0003993125

f'(0.05) = 3(0.05²) - 0.33(0.05) = 0.00585

x₁ = 0.05 - (0.0003993125 / 0.00585) = 0.0317672

Second iteration:

f(0.0317672) = 0.0000321883

f'(0.0317672) = 0.0023674

x₂ = 0.0317672 - (0.0000321883 / 0.0023674) = 0.0281852

Third iteration:

f(0.0281852) = 0.0000002655

f'(0.0281852) = 0.0018627

x₃ = 0.0281852 - (0.0000002655 / 0.0018627)

= 0.0281709

Error = |(xₙ - xₙ₋₁) / xₙ| × 100%

After 1st iteration: |(0.0317672 - 0.05) / 0.0317672| × 100% = 57.39%

After 2nd iteration: |(0.0281852 - 0.0317672) / 0.0281852| × 100% = 12.71%

After 3rd iteration: |(0.0281709 - 0.0281852) / 0.0281709| × 100% = 0.05%

import numpy as np

import matplotlib.pyplot as plt

def analyzesignalfft():

f1, f2 = 50, 120

fs = 1000

t = np.linspace(0, 1, fs, endpoint=False)

signal = np.sin(2 \* np.pi \* f1 \* t) + np.sin(2 \* np.pi \* f2 \* t)

fft\_result = np.fft.fft(signal)

freqs = np.fft.fftfreq(len(t), 1 / fs)

plt.figure(figsize=(12, 6))

plt.plot(freqs[: fs // 2], np.abs(fft\_result)[: fs // 2])

plt.xlabel("Frequency (Hz)")

plt.ylabel("Magnitude")

plt.title("FFT of the Signal")

plt.xlim(0, 150)

plt.grid(True)

plt.savefig("./assets/e-signal\_fft.png")

analyzesignalfft()

for n = 1:5

# In each iteration:

# 1. It calculates x as n \* 0.1

x = n\*0.1;

# 2. It calls a function myfunc2 with arguments x, 2, 3, and 7

z = myfunc2(x,2,3,7);

# 3. It prints the values of x and z in a formatted string

fprintf('x = %4.2f f(x) = %8.4f \r',x,z)

# The loop ends

# - The output will show 5 lines, each with different values of x and z

# - x will take values 0.1, 0.2, 0.3, 0.4, and 0.5

# - z will depend on how myfunc2 is defined

**h)**

x = [1 2 3 4 5 6];

This creates a vector x with values from 1 to 6.

y = [5.5 43.1 128 290.7 498.4 978.67];

This creates a vector y with the given values.

p = polyfit(x,y,4);

This fits a 4th degree polynomial to the data points (x,y).

x2 = 1:.1:6;

This creates a new vector x2 with values from 1 to 6 in steps of 0.1.

y2 = polyval(p,x2);

This evaluates the fitted polynomial at the points in x2.

plot(x,y,'o',x2,y2)

This plots the original data points (x,y) as circles ('o') and the fitted curve (x2,y2) as a line.

grid on

This adds a grid to the plot.

The output of this code will be a graph showing:

The original data points (1,5.5), (2,43.1), (3,128), (4,290.7), (5,498.4), and (6,978.67) plotted as circles.

A smooth curve representing the 4th degree polynomial fit to these points.

A grid overlay on the graph.

import numpy as np

import matplotlib.pyplot as plt

x = np.array([1, 2, 3, 4])

y = np.array([1, 4, 9, 16])

def lagrangeinterpolation(x, y):

def L(x, i):

L = np.ones\_like(x)

for j in range(len(x\_data)):

if i != j:

L \*= (x - x\_data[j]) / (x\_data[i] - x\_data[j])

return L

def P(x):

return sum(y\_data[i] \* L(x, i) for i in range(len(x\_data)))

return P

def newtondivideddifference(x, y):

n = len(x)

coef = np.zeros([n, n])

coef[:, 0] = y

for j in range(1, n):

for i in range(n - j):

coef[i][j] = (coef[i + 1][j - 1] - coef[i][j - 1]) / (x[i + j] - x[i])

def P(x\_val):

n = len(x\_data) - 1

p = coef[0][0]

for i in range(1, n + 1):

term = coef[0][i]

for j in range(i):

term \*= x\_val - x\_data[j]

p += term

return p

return P, coef[0]

x\_data, y\_data = x, y

P\_lagrange = lagrangeinterpolation(x\_data, y\_data)

P\_newton, coef\_newton = newtondivideddifference(x\_data, y\_data)

x\_plot = np.linspace(0, 5, 100)

y\_lagrange = [P\_lagrange(xi) for xi in x\_plot]

y\_newton = [P\_newton(xi) for xi in x\_plot]

plt.figure(figsize=(10, 6))

plt.scatter(x\_data, y\_data, color="red", label="Data points")

plt.plot(x\_plot, y\_lagrange, label="Lagrange Polynomial")

plt.plot(x\_plot, y\_newton, "--", label="Newton's Polynomial")

plt.legend()

plt.title("Comparison of Lagrange and Newton Interpolation")

plt.xlabel("x")

plt.ylabel("y")

plt.grid(True)

plt.savefig("/assets/i-Newton-Interpolation.png")

print("Newton's Divided Difference Coefficients:", coef\_newton)

x\_new = 2.5

print(f"\nValue at x = {x\_new}:")

print(f"Lagrange: {P\_lagrange(x\_new)}")

print(f"Newton: {P\_newton(x\_new)}")

**j)**

import numpy as np

def power\_iteration(A, num\_iterations=1000, tolerance=1e-8):

n = A.shape[0]

v = np.random.rand(n)

v = v / np.linalg.norm(v)

for \_ in range(num\_iterations):

Av = A @ v

eigenvalue = v.T @ Av

new\_v = Av / np.linalg.norm(Av)

if np.allclose(v, new\_v, rtol=tolerance):

break

v = new\_v

return eigenvalue, v

A = np.array([[4, 1, 1], [1, 3, -1], [1, -1, 2]])

eigenvalue, eigenvector = power\_iteration(A)

print("Dominant eigenvalue:", eigenvalue)

print("Corresponding eigenvector:", eigenvector)

def qr\_algorithm(A, num\_iterations=1000):

n = A.shape[0]

Q = np.eye(n)

for \_ in range(num\_iterations):

Q\_k, R\_k = np.linalg.qr(A)

A = R\_k @ Q\_k

Q = Q @ Q\_k

eigenvalues = np.diag(A)

eigenvectors = Q

return eigenvalues, eigenvectors

A = np.array([[4, 1, 1], [1, 3, -1], [1, -1, 2]])

eigenvalues, eigenvectors = qr\_algorithm(A)

print("Eigenvalues:", eigenvalues)

print("Eigenvectors:")

print(eigenvectors)

**k)**

import numpy as np

def f(x, y):

return x\*\*2 + y\*\*2 - x \* y + x - y + 1

def gradientf(x, y):

dx = 2 \* x - y + 1

dy = 2 \* y - x - 1

return np.array([dx, dy])

def gradientdescent(learning\_rate=0.1, num\_iterations=1000, tolerance=1e-6):

x, y = 0, 0 # Starting point (0, 0)

for \_ in range(num\_iterations):

grad = gradientf(x, y)

new\_x = x - learning\_rate \* grad[0]

new\_y = y - learning\_rate \* grad[1]

if np.abs(f(new\_x, new\_y) - f(x, y)) < tolerance:

break

x, y = new\_x, new\_y

return x, y, f(x, y)

x\_min, y\_min, f\_min = gradientdescent()

print(f"Minimum found at x = {x\_min}, y = {y\_min}")

print(f"Minimum value of f(x, y) = {f\_min}")